ECSE 426 - Microprocessor Systems
Final Project - Group 4

Matthew Lesko	Marcel Morin	Fabrice Normandin	Harley Wiltzer
260692352	260605670	260636800	260690006

April 23, 2018

Contents

List of Figures

List of Tables

1 Problem Statement

This text presents the design of a system whose ultimate goal is to gather data using the peripherals of a microcontroller and publish this data to a webserver via a chain of different communication media. From the user's perspective, the only setup required involves installing an Android application (designed for the purposes of this project) and launching a web server to collect the recorded data. After the Android device has paired with the Nucleo microcontroller, the data recorded from the accelerometer or microphone attached to the STM32F407 Discovery board will be transmitted automatically to the webserver, where the data may be visualized numerically and with pictorial support. If the data being sent is a microphone recording, the web server will perform *speech recognition* to parse a number n from the recorded audio, which will be communicated all the way back to the Discovery board, causing an LED to blink n times.

Given the reduced amount of resources usually present in microcontrollers, several devices were required to work in tandem in order to create a pipline from the Discovery board to the Internet. The Discovery board, while having access to an accelerometer and an ADC for reading microphone data, has no resources for communicating over Bluetooth or HTTP. Instead, it may communicated via a wired connection to the Nucleo board that is fitted with a Bluetooth Low Energy (BLE) transmitter. Given that the Nucleo has no WiFi capabilities, it communicates data over BLE to an Android application, which may then tunnel the data over HTTP to the web server. Thus, in order to communicate sensor data from the Discovery board all the way to the web server, three distinct communication protocols are used (UART, BLE, HTTP), each having their own unique quirks and *impedimentum*. A major challenge of the design of this system revolved around the design of how to encapsulate data to be sent, and how to ensure the timing between transmitters and receivers was appropriate.

Furthermore, given the plethora of devices responsible for the transmission of data in this system, it was necessary to implement four distinct software components across three different technologies. As will be described further in this report, both the Discovery and Nucleo board required independent embedded-C programs. Furthermore, to bridge the gap between the microcontrollers and the web application, an Android app was devised and implemented in Java. Finally, the web application prompted the design of a RESTful API as well as a fancy interface for displaying the data it received. This was achieved mainly with the Python language and some of its excellent libraries.

Finally, another design challenge was that of speech recognition. Although a custom Convolutional Neural Network was initially designed for this task, the amount of time that would have been required to train it properly would have severely hindered progress on the rest of the project. Therefore, the speech recognition feature was outsourced to a Google Speech API.

The remainder of this text will describe how the system was designed, and will discuss the various challenges were experienced as well as how they were resolved. Ultimately, the system proposed in this paper is meant to show how embedded systems can communicate with each other, as well as how they may communicate over the vast Internet. Thus, the proposed system exhibits the *Internet of Things*, a field and concept that has been growing tremendously and is projected to grow into a market worth $7.1 trillion by the year 2020[**?**].

The legendary philosopher John Milton once said "give me the liberty to know, to utter, and to argue freely according to conscience, above all liberties"[**?**]. The Internet of Things and the design proposed in this paper ultimately recognize microcontrollers as entities capable of knowing, uttering, and arguing, which was, above all else, demonstrated by the proposed system. The Internet of Things revolution, therefore, shall eventually bring the "Inter-computational Covenant on Civil and Political Rights", allowing all devices to communicate equally to advance society. The authors hope that the results demonstrated in this text show promise in the evolution and advancements in the freedom of device-expression.

2 Theory

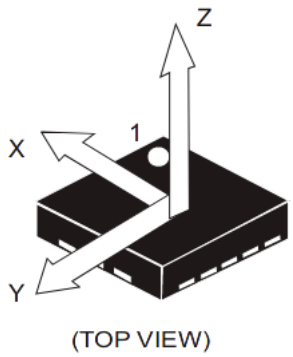
2.1 The Accelerometer

The STM32F407 Discovery board is equipped with an accelerometer which measures acceleration on three axes. The accelerometer is a peripheral that communicates with the Discovery's CPU via the Serial Peripheral Interface (SPI), where the Discovery is configured as Master and the accelerometer peripheral is configured as Slave. Since the Discovery does not transmit anything to the accelerometer, only its SPI RX line is in use, whereas only the SPI TX line is used on the accelerometer. The Discovery and the accelerometer share a clock, generated by the Discovery's CPU (because it is configured as the Master device), and share a common ground.

The directions over which accelerations are measured by the device are shown in Figure 1.

However, for the purposes of this project, the goal was to transform the raw accelerometer

Figure 1: Accelerometer measurement basis[1]



data into more symbolic quantities. As such, the *pitch* and *roll* measurements were of interest. These quantities are often associated with the orientation in 3D space of airplanes, for example. Conventionally, the pitch is defined as the rotation about the x axis, and the roll is defined as the rotation about the y axis. Thus, letting a_x, a_y, a_z be the acceleration measurements along the x, y, and z axes respectively, we may compute pitch as follows:

$$\mathsf{Pitch}(a_x, a_y, a_z) = \left(\frac{180}{\pi}\right) \arctan \frac{a_x}{\sqrt{a_y^2 + a_z^2}} \tag{1}$$

where the $180/\pi$ term scales the measurement to degrees from radians. Similarly, we compute the roll as follows:

$$\mathsf{Roll}(a_x, a_y, a_z) = \left(\frac{180}{\pi}\right) \arctan \frac{a_y}{\sqrt{a_x^2 + a_z^2}} \tag{2}$$

2.2 Communication Between the Two Microcontrollers: UART

In order to transmit microphone or accelerometer data to eventually reach the web server, it must be communicated from the Discovery Board to the Nucleo Board, since the Nucleo Board has the

[1] http://www.st.com/resource/en/datasheet/lis3dsh.pdf

ability to do Bluetooth Low Energy communication. In order to communicate data between the Discovery and the Nucleo, the Universal Asynchronous Receiver/Transmitter (UART) protocol is used. UART is a very simple system that allows communication between two devices wired together. Each device requires a ground pin (`GND`), a transmission pin (`TX`), and a receiver pin (`RX`). Since the communication is asynchronous, no clock or synchronization lines are required. To distinguish when a transmission has ceased or begun, special start and end sequences of serial bits are standardized. Given both the `RX` and `TX` lines, however, full duplex communication is possible. However, for the purposes of this system, full duplex communication was not necessary. However, bidirectional communication was implemented.

Using the HAL drivers, communication via UART is very simple. The `HAL_UART_Transmit` function merely takes a handle to the UART device, a buffer of data, the length of the buffer, and a timeout period, and transmits the data in the buffer serially on the `TX` line. Similarly, the `HAL_UART_Receive` function takes a handle to the UART device, a buffer for data, the amount n of bytes to receive, and a timeout period, and fills the buffer with the data read serially from the `RX` line. This function call is *blocking*, meaning the program waits for all n bytes to be received before proceeding. Moreover, the `HAL_UART_Receive_IT` function was used, which accomplishes the same task as `HAL_UART_Receive`. However, this function is not blocking, and instead triggers an interrupt when the n bytes have been received.

An example of how to connect the Discovery and Nucleo boards for proper UART communication is given in Figure 3.

2.3 Bluetooth Low Energy

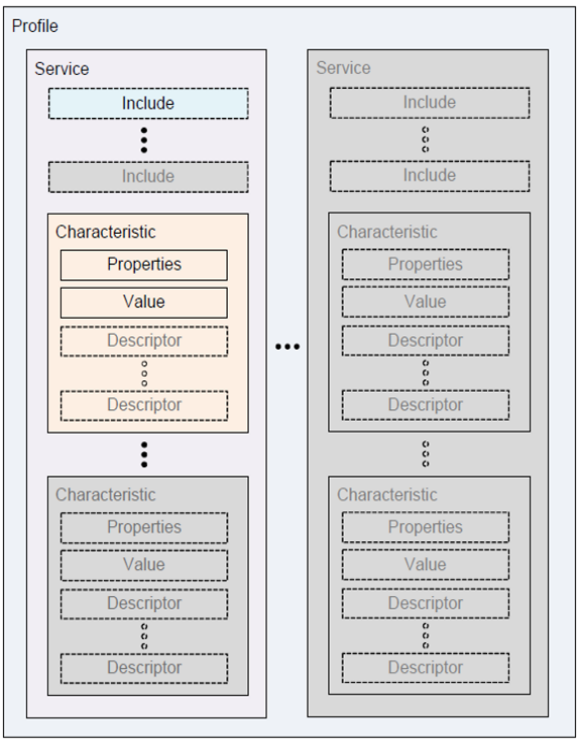
Bluetooth low energy (or BLE for short) is a wireless personal area network technology aimed at applications in health care, fitness, security, and home entertainment. It allows peripheral devices such as smart phones, medical (heart rate, pressure) sensors and wireless earphones to transmit data within a personal area network. Compared to the classic Bluetooth, its purpose is to provide reduced power consumption while maintaining a similar range of communication. It saves power by sending small bursts of data as opposed to a continuous stream as used in classic Bluetooth technologies. This section of the report details the theory behind BLE and its use cases relevant to the project [**?**].

The purpose of BLE in this project is for it to allow the pair of Nucleo and Discovery boards to communicate data over BLE to a smart phone which then transmits that data to the internet. BLE works similarly to Bluetooth, in which a *Central Device* scans for *Peripheral Devices* which *advertise* their personal area networks. At this point, once the central and peripheral device connect to one another, one acts as the *Client Device* which receives data from the *Server Device*.

For sending data, the server does so by packaging its data within a *Value* under a *Characterstic*, each of which are packaged within a *Service*, this hierarchy of data is part of the *GATT Profile Hierarchy*.
One can see this hierarchy within the following Figure 2 [**?**].

Each server device has only one GATT profile, but within the profile, it may have multiple services, which represent different features. For example, one service can be for a heart rate sensor,

Figure 2: GATT Profile Hierarchy



and another for a pressure sensor. In this project, there is only one custom made service. A service may have multiple characteristics, which represents different data. For example, one characteristic can for a heart rate measurement and one can be for configuration settings for the heart rate sensor. Within each characteristic, there is a *Value*, the actual data. Then there contains one or many *Descriptors*, which describe a value. Finally, *Properties* contain the read, write and notify permissions for that characteristic values.

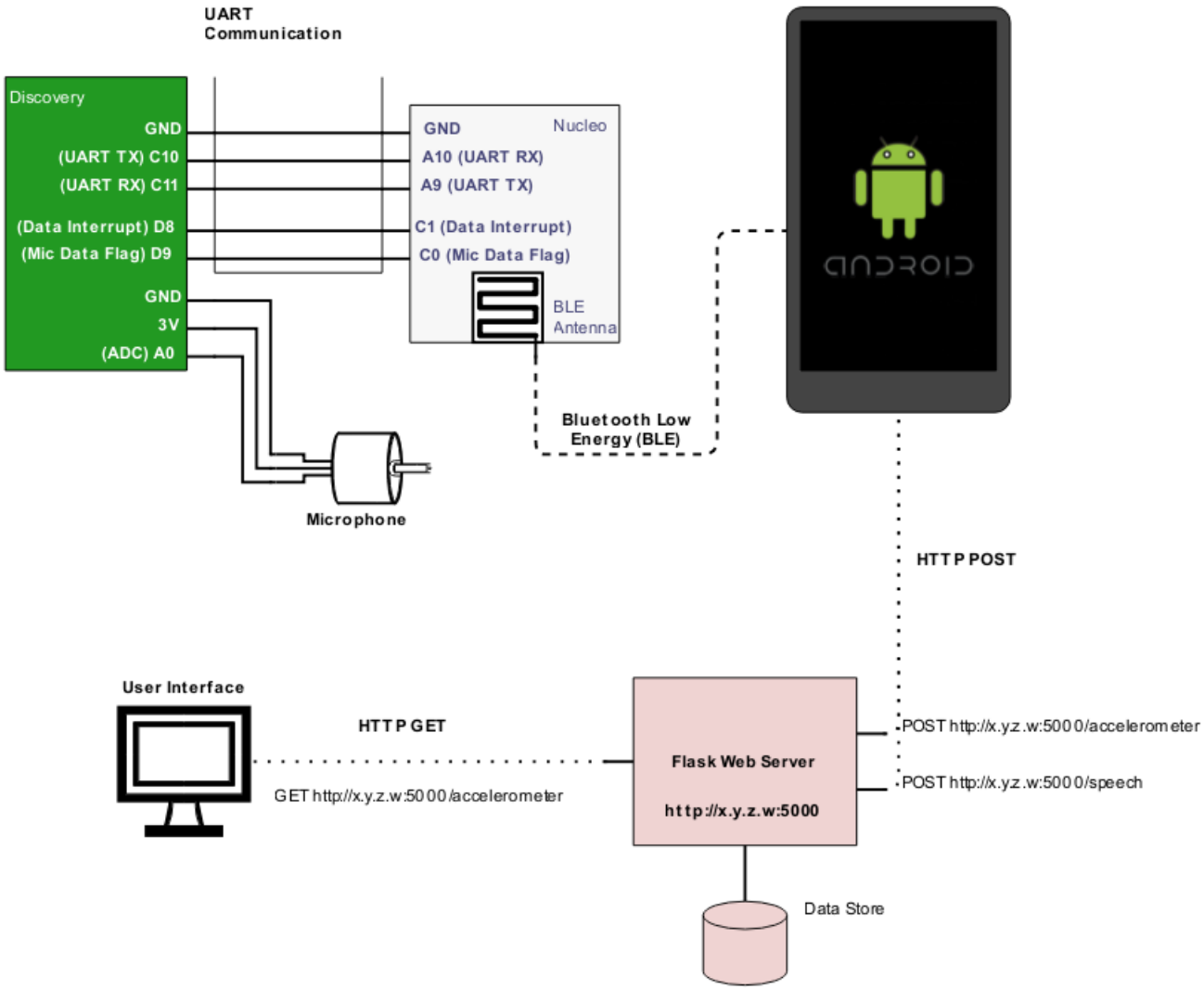
3 Implementation

3.1 An Overview of the System Architecture

Figure 3 shows the high-level architecture involving all parts of the system described in this paper. The pipeline begins at the Discovery board, where the user tapping the board prompts data to be recorded. This data is then transmitted over UART to the Nucleo board, which has the ability to transmit data via Bluetooth Low Energy. The Discovery and Nucleo blocks in Figure 3 explicitly demonstrate how to connect the two for proper UART communication.

The Android application discussed in this paper automatically connects to the Nucleo Board

Figure 3: Architecture and Connectivity of the Overall System



when the "Scan for Nucleo" button is pressed. Then, the app should receive the GATT ser-

vices and characteristics broadcast by the Nucleo, allowing for BLE communication to proceed.

With the Android app configured to send to the appropriate IP address where the Flask web server is hosted (that is, at `http://x.y.z.w:5000` in the example in Figure 3), it will automatically send the data it receives over BLE to the right endpoint of the web server. In the event that the Android app communicates microphone data, it will then wait for an HTTP response for the web server containing the perceived digit recognized from the audio samples, and the response will be sent backward through the pipeline until it eventually reaches the Discovery Board.

Upon receiving an `HTTP POST` at the `/accelerometer` endpoint, the web server will parse the accelerometer data from the body of the request, save the data in a CSV file on the server, and generate a graph of the data. Then, when the user interface is requested via `HTTP GET`, the graph and the CSV data are presented on a web page.

Upon receiving an `HTTP POST` at the `/speech` endpoint, the web server will extract the WAV file from the request and use Google Cloud's Speech API to decipher a digit from the audio file. The digit that it perceives is then sent back to the sender of the audio data via an HTTP response.

When the digit response containing some integer n finds its way back to the Discovery board, the Discovery's blue LED blinks n times. At this stage, the process starts over, where the Discovery waits for user input.

The remainder of this section discusses the implementation and functionality of all parts of the system in more detail.

3.2 The Beginning of the Pipeline: The STM32 Discovery Board

The STM32F407 Discovery Board is the microcontroller that the user interacts with. As discussed previously, this board is responsible for gathering accelerometer and microphone data to send through the pipeline. Figure 4 demonstrates the Finite State Machine that the Discovery Board adheres to.

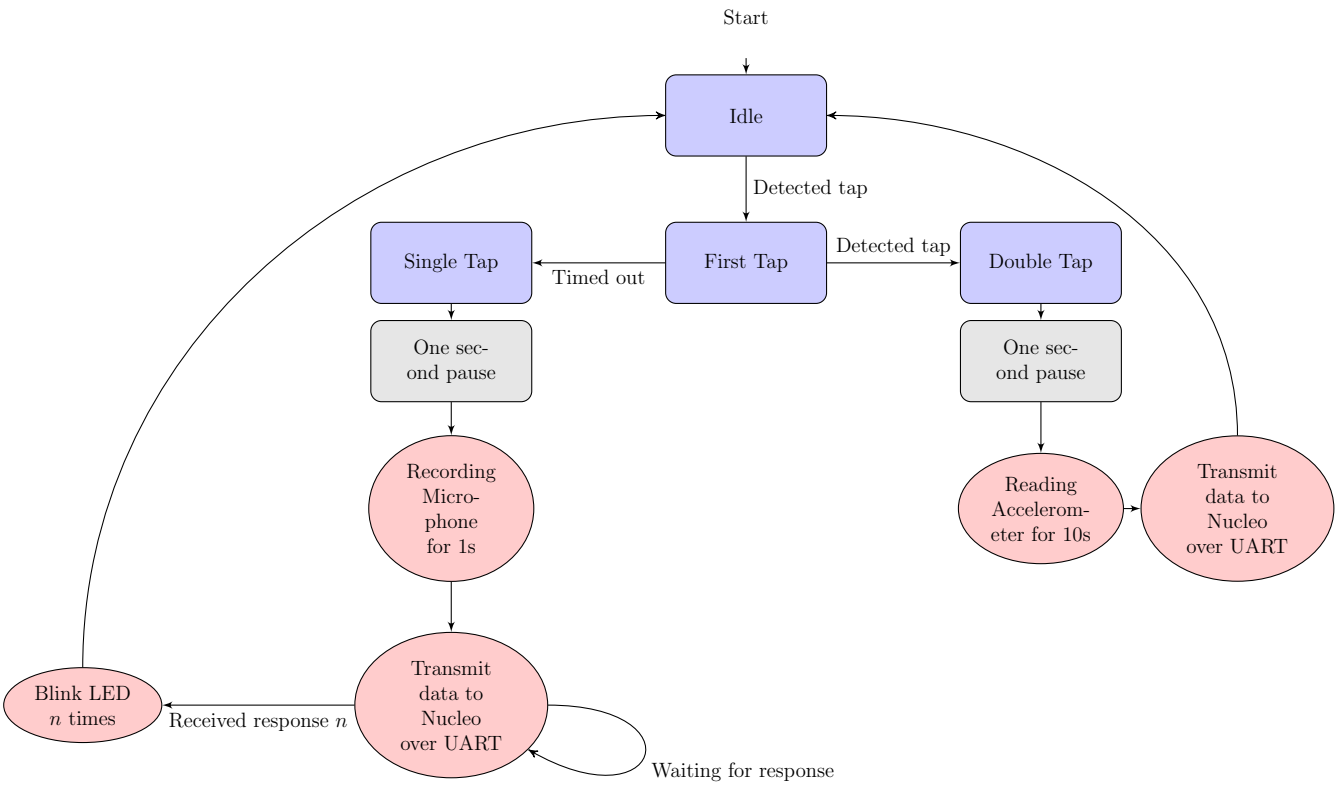
The system starts in the Idle state, where it waits for user input in the form of taps on the board itself. The taps are detected by accelerometer readings polled at 100Hz over 500ms intervals, when an accelerometer reading on the z axis surpasses the average reading over the window by a certain threshold. In order to filter out noisy accelerometer readings, an exponential moving average filter is applied, which filters as follows:

$$\hat{z}_t = \alpha z_t + (1 - \alpha)\hat{z}_{t-1}, \alpha \in (0, 1] \tag{3}$$

where z_t is the tth accelerometer reading, and $\hat{z}$ is tth output of the filter. The parameter α was chosen through testing, and it represents how "smooth" the filter output should be.

When a tap is detected, the system enters the First Tap state, and the green LED is illuminated for visual feedback, so the user can gauge how rapidly to make a double tap. If no tap tap is

Figure 4: FSM on the Discovery Board



detected in the next 500ms window, the system enters the Single Tap state (denoting that it has detected only a single tap) and the blue LED is illuminated, and otherwise, it enters the Double Tap state where the orange LED is illuminated. These states ultimately determine which data to record, and how the rest of the system proceeds. In either case, there is a one second delay after the transition to these states to allow the user to prepare for the recording.

3.2.1 Sending Microphone Data

In the event that the system enters the Single Tap state, after the aforementioned one second grace period, the machine will begin to record data from the microphone. This data is sampled at 10kHz by the onboard analog to digital converter (ADC) for 1 second using direct memory access (DMA). During this recording period, the Discovery Board's red LED will be blinking. When the blinking stops, the recording has finished. In order to retain high audio quality, the ADC converts samples to 12-bit integers. Since the FSM is not in use during the recording period, it begins to wait on a signal before continuing. This effectively prohibits the FSM from being scheduled by the CPU until the signal is set. Once the DMA buffer is filled with ADC readings, a callback function is triggered, which in turn sets the signal allowing the FSM to resume.

After receiving the signal from the ADC DMA callback, the FSM can once again be scheduled.

However, before sending the microphone data to the Nucleo board, it is desired to *compress* the data returned by the DMA process. Since DMA buffers contain 32-bit words and ADC samples are only 12 bits wide, it is desirable to *squash* the data in the DMA buffer into 16-bit integers. This was done by an elegant `squash` function, that performs this operation in place. This function can be seen in Figure 5. This halves the amount of bytes required to be sent to the Nucleo board over UART.

```
/** Squashes an uint32_t array down to uint16_t in-place. */
void squash(uint32_t array[], int length){
 // Create two pointers, pointing at the start of the array.
 uint32_t * source = &array[0];
 uint16_t * destination = (uint16_t*) &array[0];
 for (int i=0; i<length; i++, source++, destination++){
  // We copy from source -> destination, within array.
  *destination = (uint16_t) *source;
 }
}
```

Figure 5: Neat little *squash()* function.

With the compressed data ready, the Discovery Board asserts a `Data Interrupt` GPIO pin, which is directly connected to the Nucleo Board. This will trigger an interrupt on the Nucleo, essentially informing it that it should be ready to receive a UART transmission. Furthermore, it asserts a `Mic Data` GPIO pin, also connected directly to the Nucleo Board, which tells the Nucleo that it is about to receive microphone data. Then, the Discovery board begins transmitting the microphone data over UART.

Once the UART transmission is complete, the Discovery Board must wait from a response indicating how many times its LED should blink as indicated by the microphone reading. Since the amount of time necessary to receive the response is relatively enormous, it is undesirable for the Discovery Board to poll for UART responses. Thus, it initiates a UART Receive in Interrupt Mode, which will trigger an interrupt when the desired byte is received from the Nucleo Board. Meanwhile, the FSM waits on another signal in order to prevent it from being scheduled by the CPU. This signal is set when the UART Receive callback is triggered.

Upon receiving a response from the Nucleo Board, the discovery extracts a number $n \in \{0, 1, \ldots, 9\}$ from its receive buffer. Then, the blue LED will blink n times, indicated the number that was recognized from the initial microphone reading. After the LED has finished blinking, the system returns to the Idle state, ready to start all over again. At this stage, no LED's should be illuminated.

3.2.2 Sending Accelerometer Data

In the event that the system enters the Double Tap state, after the aforementioned one second grace period, the machine will begin to record accelerometer readings at 100Hz for 10 seconds. As in the case of the microphone recording session, the red LED will blink for the duration of the

accelerometer recording. In order to reduce the amount of data that will be sent over UART (and later, over BLE), the Discovery Board converts each accelerometer reading (x_i, y_i, z_i) into pairs (p_i, r_i), where p_i and r_i are the corresponding pitch and roll measurements, respectively. Then, rather than sending $3*1000 = 3000$ floating point numbers, only $2*1000 = 2000$ floats must be sent.

Note that in contrast to the case of the microphone data, where it was desired to compress the readings into 16-bit integers, the accelerometer data should be kept at pairs of 32-bit floats to retain enough precision. However, before transmitting the data it is passed through an exponential moving average filter. However, this filter uses a less smooth α parameter as that which was used for detecting taps – this is due to the fact that it was easier to detect taps when the accelerometer readings were smoothed out as much as possible. To keep the integrity of the accelerometer readings, less filtering was done for the recorded data.

Once the Discovery Board has finished filtering all 10000 pairs, it is ready to transmit the data over UART to the Nucleo Board. As in the case with microphone data, the `Data Interrupt` pin is asserted. However, in this scenario, the `Mic Data` GPIO pin is reset – this way, the Nucleo knows that it is about to receive accelerometer data. Next, the pitch and roll are sent in consecutive UART transmissions. The Discovery first transmits the buffer of pitch data, and waits 10ms before transmitting the roll data. The 10ms delay allows time for the Nucleo Board to prepare for another UART reception.

After transmitting the accelerometer data, it will eventually be uploaded to the Web Server implemented for this project. However, in the case of accelerometer data transmission, the Discovery Board *does not* get a response. Thus, it returns immediately to the Idle state, where the system is ready to start all over again. At this point, no LED's should be illuminated.

3.3 The Nucleo Board

The Nucleo board acts as an intermediate between the Discovery board and the Phone. Its main role is to relay information between these two devices, using the appropriate communication protocols, and by correctly serializing and de-serializing data.

3.3.1 UART Communication with the Discovery Board

As described previously, the sampled data is sent over UART after each data collection period. In order to differentiate between accelerometer and microphone samples, a GPIO pin (with the alias `IS_MIC_DATA`) is used. When set, the data received via UART represents microphone samples, and vice versa for accelerometer samples. Additionally, another GPIO pin, `DATA_INTERRUPT`, was configured in rising-edge triggered interrupt mode, and was used to coordinate data transfers between the Discovery and Nucleo boards.

The GPIO Interrupt-Service-Routine, when triggered from the `DATA_INTERRUPT` GPIO pin, sets a conditional flag, which is then read inside the main loop of the program body, and allows one execution of the `pipeline()` function before being reset. This `pipeline()` function contains the main logic of the Nucleo program. Its logic is fairly straightforward, and can be viewed in the diagram of Figure 6.

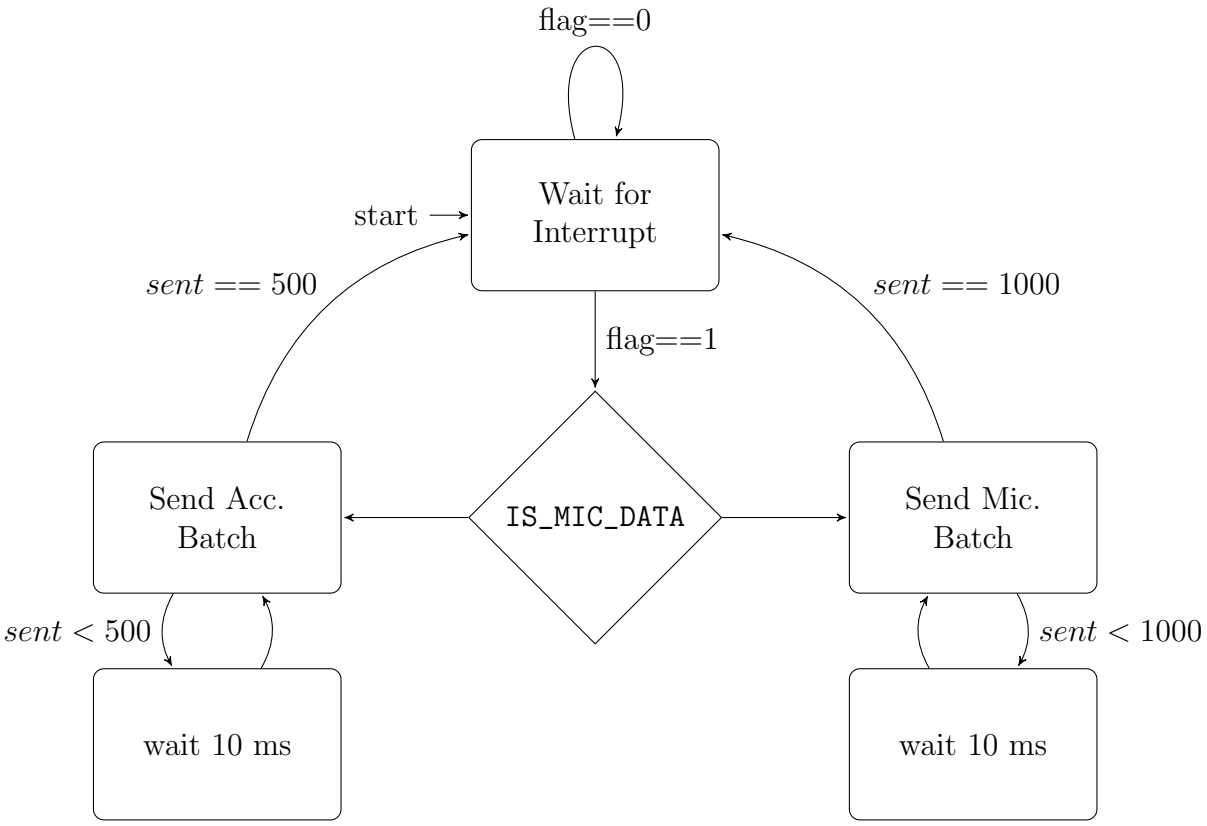


Figure 6: Conceptual State-Diagram of the Nucleo board's `pipeline()` function.

3.3.2 Bluetooth Low Energy Communication with Phone

In order to reliably transmit data over BLE, we have to be mindful of the throughput limitations associated with this protocol. Following some quick research, it was determined that the maximum application data that could be transmitted for a given attribute was 20 bytes[**?**]. Given this constraint, we devised a serialization scheme for both accelerometer and microphone samples. In the case of accelerometer data, the 1000 samples of pitch and roll are interleaved, and then split into batches of two pairs of samples. This effectively creates packets containing 4 float values, or 16 bytes of usable data in total. We decided to opt for an even number of samples, even though another float value (4 bytes) could also have been added, in order to simplify the code. An illustration of the serialized samples can be seen in Figure 7.

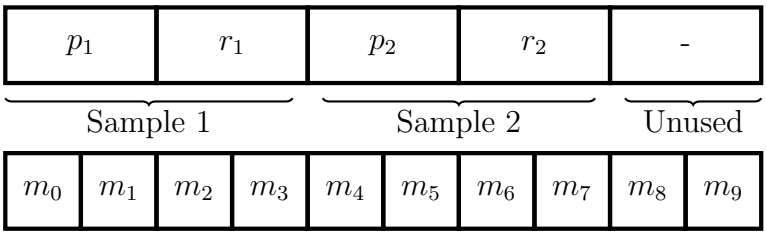


Figure 7: Accelerometer and Microphone data packets structure.

As will be further discussed later in BLE Interface, the Nucleo board and android application communicate using the event broadcasted when a GATT attribute changes. As such, we chose

to opt for simplicity and reuse most of the infrastructure of the BLE Sample Project, which was provided on *MyCourses*, and implemented a `CustomService`, containing our own implementations of methods for adding the custom service, the three characteristics (Microphone, Accelerometer, Digit), as well as the `GATT_attribute_changed()` callback. The serialization and deserialization logic in both the Android application and Nucleo boards behaves identically, with Big-Endian representation used in each case to facilitate encoding and decoding, as well as debugging.

3.4 Smart Phone Android Application

Given that the Nucleo board has no WiFi/Internet capabilities, data must be communicated over BLE to an Android application, that then forwards the data to the web server over HTTP. For convention, this report may use "Android app" to signify the smart phone BLE mobile application. The report also assumes that the reader has basic knowledge of telecommunication protocols, and Android applications. This portion of the report details the implementation and the rationale behind the design of the smart phone application that acts as the intermediary component between the Nucleo board and the web server.
With that being said, the requirements for the Android application are the following:

- Scan for and connect to BLE peripheral devices;
- Enable the user the ability to start, or stop scanning for devices, and connect or disconnect from one peripheral device;
- Discover BLE services and characteristics from the peripheral device;
- Read voice and accelerometer data batches over BLE from the Nucleo board;
- Save the received data to its appropriate file;
- Once the accelerometer file contains 10 seconds worth of data, transmit the file to the web server over HTTP;
- Once the voice file contains 1 second worth of data, transmit the file to the web server over HTTP;
- Handle HTTP responses from the web server and send data to the Nucleo board.

Each of the Android application's business-logic features are implemented in the Java programming language, and the user interface is designed in XML. The implementation of the Android application is facilitated by using the Android Studio IDE.

3.4.1 Brief Summary of Android

The report doesn't go into detail about the code written for the project, if the reader desires to see the source code themselves, they can see it here. There is documentation written within the code so that the reader may understand the basics of the code written. This report goes into detail

about the high level designs and concepts used.

Android applications perform in a way such that all user interface and business-logic is performed within an *Activity*. Even if a developer doesn't need a user interface for their application, a main activity must instantiate and start once the user opens the application. Android developers must also declare any build dependencies such as programming libraries and hardware capabilities (such as Bluetooth and Internets) within the *manifest.xml* file. The activity, since there can be multiple activities, which runs on application start up is also declared in the manifest file.

3.4.2 BLE Interface

With regards to BLE, the smart phone is recognized as the *client device*, and the Nucleo board as the *server device*.

The first step for any BLE handshaking, is the *Scanning Phase*, which is facilitated by a *BLE Scan Callback*. For the client device to recognize BLE servers, the server must advertise itself and the client must scan for these advertisements. With the push of a *SCAN* button on the Android app user interface, the application enters a scanning mode which, by the help of a *Scan Callback* Java class, saves scanning results as MAC addresses that the Android application could try to connect to. Since scanning is cumbersome on the smart phone's battery, the scanning automatically stops after 10 seconds to save battery life. To facilitate full automation, the Android app automatically stops scanning once it finds the Nucleo board's MAC address and tries to connect to the board. This improves the time required to enable a connection between the client and server since the user doesn't have to look through a list of MAC addresses to find the Nucleo board's address and have to select that specific address.

The last phase for the BLE handsaking is the connection phase, which is facilitated by a *GATT Client Callback*, where GATT stands for *General Attribute*. Once the Android app finds the Nucleo board, it automatically tries to connect to it, the GATT Client Callback takes charge of this operation. On the press of a CONNECT toggle button, the user can decide to connect or disconnect from the selected peripheral device.

The GATT Client Callback handles any connection events, and outputs statuses such as *Connection Success* and *Connection Failure*. Since there isn't any authentication procedure between client and server devices, the GATT Client Callback simply notifies the server that it wishes to connect to it and once the server, by the help of a GATT Server Callback, receives the notification, and responds with a connection success. If somehow the connection is a failure, the Android application simply logs the result to the Android Studio debug console. Although, on a connection success, the Android application now starts to search for *Services* that the server may be transmitting. If the service used to package voice and accelerometer data is discovered, and contains their appropriate *Characteristics*, the Android application is ready to read and be notified by any updates to the voice and accelerometer data.

At this point, the smart phone is connected to the Nucleo board over a BLE session and can now start to transmit data to one another! Exciting, if I do say so myself.

3.4.3 Handling Data

The Android app receives data in a byte array which is packaged under a characteristic assigned for either voice or accelerometer data. The batch of data is then written to its appropriate file with the help of the `AppController.java` class.

To send files over HTTP, the Android app makes use of the Volley Library to create and configure HTTP requests, while also handling any HTTP responses. This is useful if the developers wish to send files conveniently over HTTP as well as receive data from the web server over HTTP. The developers wished to keep data uploading simple. By encoding the file-to-be-sent into a base-64 string and adding the encoding to the HTTP request's form, the web server can easily retrieve the file from the request by reading and decoding the form body.

If the web server reads the voice data file, then the Android application receives a response containing the output from the Google Speech API, under the parameter "reading". This contains the number that the Google Speech API was able to identify given the voice data. The Android application sends this data to the Nucleo board over BLE by writing the value to its appropriate Characteristic. At this point, the data is now handled by the Nucleo server.

3.4.4 Summary of Project, Structure and Classes

One can see a visualization of the Android app project structure in Figure 8. Of course there are many files related to the application that couldn't be referenced under this report due to sizing constraints, any three dots represents that.

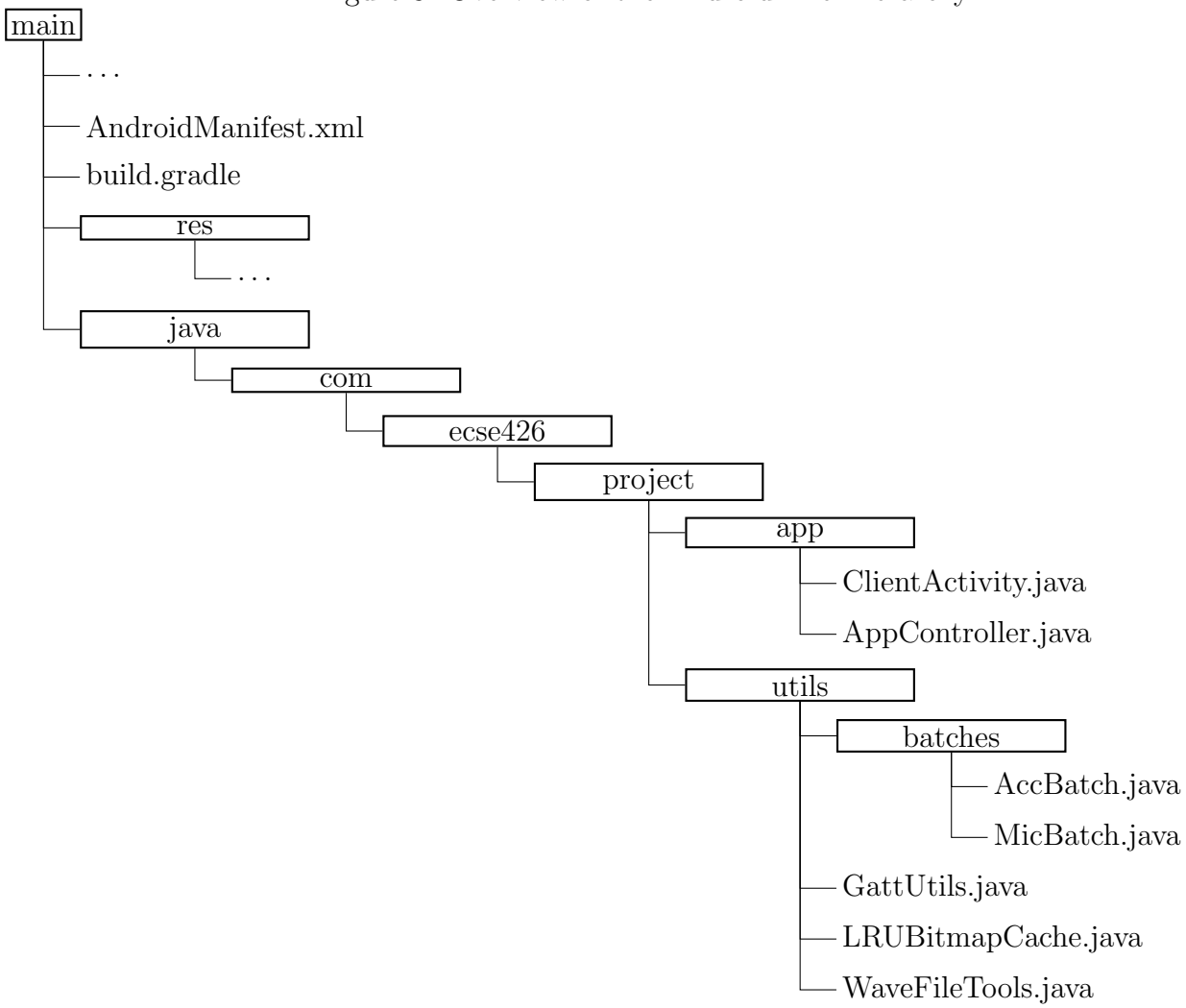
The `AndroidManifest.xml` is the manifest file specific to this application which contains dependencies. The `build.gradle` file is the gradle build script, Android Studio automatically runs its build configurations whenever the developer runs or builds the application.

The `ClientActivity.java` class is the main activity that runs on start up. It is the core of the application, every other class is written around it and is used to help it. This class contains the BLE Scan Callback and GATT Client Callback logic implemented under inner classes. It handles user input and contains function calls from other classes that receive and save data over BLE and HTTP.

The `AppController.java` handles HTTP requests by facilitating the Client Activity with a queue for requests. It also handles reading and writing voice and accelerometer data to their appropriate files. It is important to note that the `AppController.java` class extends the Application class, making it able to run simultaneously with the `ClientActivity.java` class and have data persist when the application is closed.

The `AccBatch.java` and `MicBatch.java` classes are helper classes that contain functions to convert accelerometer or voice data respectively from bytes to a data type instructed by the function. The `GattUtils.java` class contains GATT Service, Characteristic, and Configuration UUIDs, as well as the Nucleo board's MAC address. The `WaveFileTools.java` class handles reading and writing data from and to files of Wave format. The `LRUBitmapCache.java` class is supposed to send and receive files, unfortunately it is not used.

Figure 8: Overview of the Android File Hierarchy



Under the `res` directory, one can find all the user interface layouts and images. Any user interface design or image is found under that directory.

3.5 The End of the Pipeline: A Convenient Web Server

In order to perform more complex data processing and to have somewhere convenient to store the recorded data, the web server was implemented. This was achieved with the use of Python's Flask library for creating a minimal HTTP server with a REST API and a nice, simple interface to visualize the data that it receives.

The server has two main API endpoints, one for each of accelerometer and microphone data. Thus, if the server is running on `151.155.209.95:48080`, the accelerometer endpoint is accessed via a `POST` request to `151.155.209.95:48080/accelerometer`, and the microphone endpoint is accessed via a `POST` request to `151.155.209.95:48080/speech`.

Flask manages API endpoints with a `Resource` interface. This interface provides `get()`, `post()`, and `put()` methods, which are called when the endpoint receives the corresponding HTTP packet. Thus, in the design of the web server, the `Accelerometer` and `Speech` classes were created that inherit from the `Resource` class.

3.5.1 The Accelerometer Endpoint

The `/accelerometer` endpoint accepts `POST` requests and is responsible for preprocessing accelerometer data, storing the data on the server, and plotting a graph of the data it receives. `POST` requests to this endpoint should have JSON bodies as described in Table 1. The API system is

Table 1: JSON format for `POST` requests to the `/accelerometer` endpoint

Parameter Name	Parameter Type	Description
"accelerometer"	`[float]`	List of accelerometer readings. Pitch values at even indices.
"starttime"	`String`	String indicating the time at which the recordings were sent to the API.

configured bind the `Accelerometer Resource` to the `/accelerometer` endpoint. As such, this `Resource` implements the `post()` method to handle the accelerometer data that it is sent via `HTTP POST`.

Since the "accelerometer" member of the `POST` request accepts an array of interleaved pitch and roll values, the data needs to be processed such that the pitch and roll could be stored in a more convenient format for plotting. The first task of this method is to parse the "accelerometer" member of the `POST` request to split up the pitch and roll data into their own respective lists. During this procedure, the data is written to the `data.csv` file on the server in comma-separated value (CSV) format. Each row of the CSV file contains a pitch reading and the corresponding roll reading. Furthermore, the "starttime" member of the `POST` is extracted and stored in a global variable so it can later be inscribed on the user interface.

With the pitch and roll values split into their own lists, they can be graphed with relative ease. The powerful Matplotlib[**?**] library is used to plot the graph, and the pitch and roll data are both plotted in the same plane. The plot is saved to the `static/graph.png` file, and is then available to be rendered on the user interface.

Since the accelerometer communication is a one-way procedure, the `Accelerometer Resource` has nothing interesting to return to the device that sent the `POST` request. Thus, its response contains only an acknowledgement string.

3.5.2 The Speech Endpoint

Similarly to the accelerometer endpoint, implementation of the speech functionality on the web server required the construction of a `Speech Resource`, which was bound to the `/speech` endpoint. The files portion of the body of the `POST` requests that this endpoint expects is shown in Table 2.

With the WAV file extracted from the `POST` request, the task of the speech endpoint is to perform

Table 2: Format for `POST` request files sent to the `/speech` endpoint

Parameter Name	Parameter Type	Description
"audio"	WAV File	File containing audio recording in WAV format.

speech recognition. This is accomplished with the help of Google Cloud's Speech API. Fortunately, Google Cloud provides a Python interface for their fabulous speech recognition algorithms. The API's `transcribe_audio()` method is called on the WAV file, and a string is produced, containing the transcription of the audio recording. We try to cast this transcription to an integer, and if that fails we deduce that no number could be understood from the audio recording. In this scenario, we report a value of 0 in the `result` variable. Otherwise, `result` is assigned the number yielded by casting the transcription to `int`.

Contrarily to the case of the accelerometer endpoint, the microphone data communication process in this system is actually bidirectional. Thus, the sender of the microphone data expects a response from the web server, which will be passed all the way back to the Discovery board. The `Speech Resource`'s `post()` method returns a JSON object, which is illustrated in Table 3.

Table 3: JSON format for the response returned by the speech endpoint to `POST` requests

Parameter Name	Parameter Type	Description
"reading"	`int`	Byte containing the number transcribed from the input audio file. If no number is detected, 0 is returned.

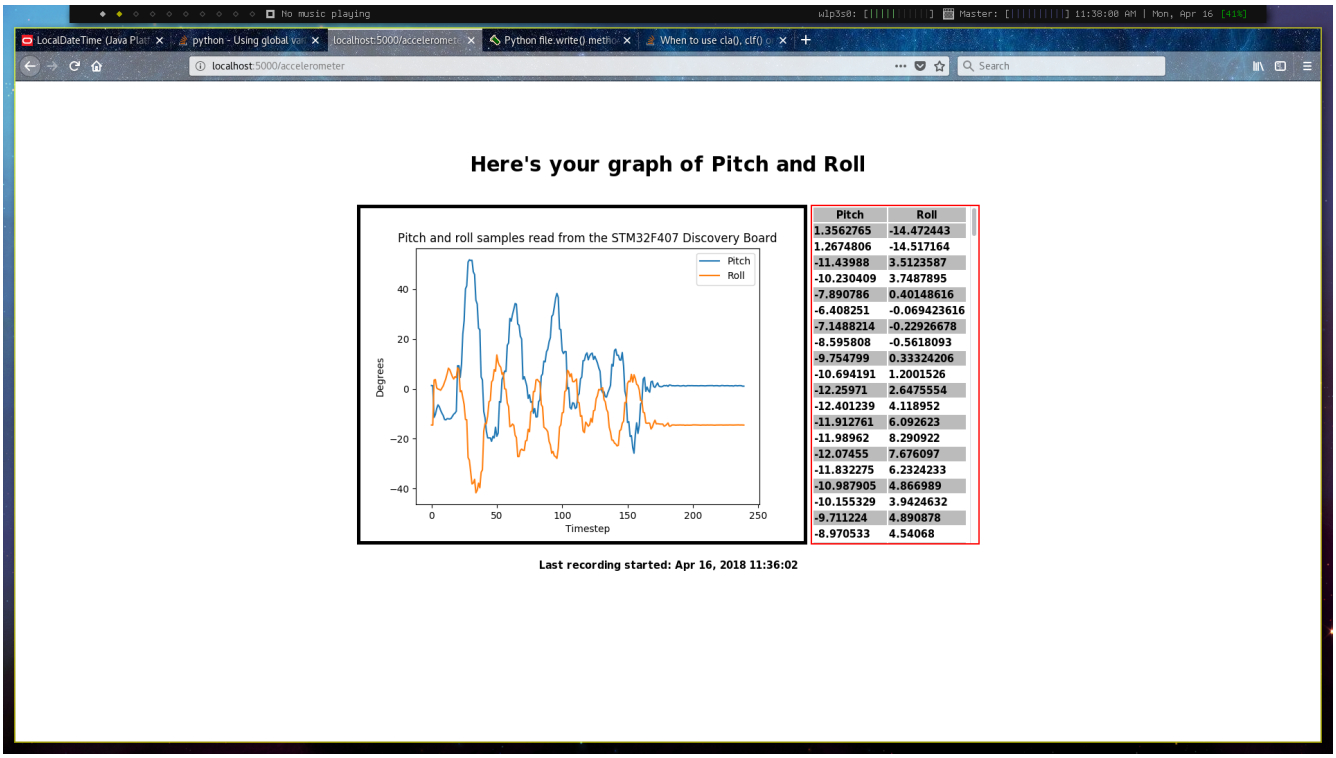
3.5.3 User Interface

A user interface was implemented in order to have a convenient way of visualizing the accelerometer data. This was designed as a web page hosted by the Flask server described above. To realize this web page, a simple HTML layout was created that exploits Flask's *template* functionality. Templates allow developers to dynamically pass parameters to HTML files – in this scenario, this was used to dynamically update the accelerometer data and recording time shown on the page.

To access the web page, whilst assuming the web server is running on `151.155.209.95:48080`, one would navigate to `151.155.209.95:48080/accelerometer` in their favorite web browser. The interface then displays the graph of the pitch and roll data, and shows the time at which the data was sent directly underneath the graph. Furthermore, the UI presents a scrolling table of the CSV data saved in `data.csv` for the user to peruse comfortably. A screenshot of this user interface is shown in Figure 9.

In order to view the latest data, one must just refresh the web page. The graph displayed

Figure 9: Screenshot of the web server's User Interface



Pitch	Roll
1.3562765	-14.472443
1.2674806	-14.517164
-11.43988	3.5123587
-10.230409	3.7487895
-7.890786	0.40148616
-6.408251	-0.069423616
-7.1488214	-0.22926678
-8.595808	-0.5618093
-9.754799	0.33324206
-10.694191	1.2001526
-12.25971	2.6475554
-12.401239	4.118952
-11.912761	6.092623
-11.98962	8.290922
-12.07455	7.676097
-11.832275	6.2324233
-10.987905	4.866989
-10.155329	3.9424632
-9.711224	4.890878
-8.970533	4.54068

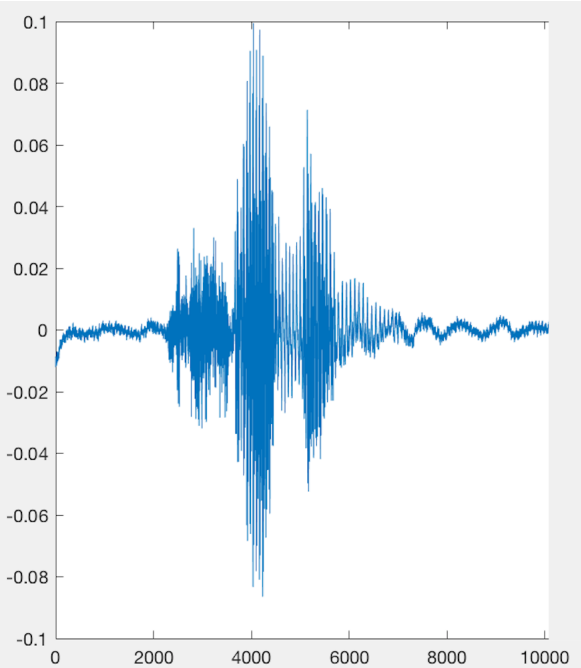
on the user interface as well as the data used to generate the graph and the table shown are persisted on the server, so they can be viewed offline as well.

4 Testing and Observations

4.1 Microphone/ADC

Regarding microphone data, there were two aspects of this component that needed to be tested. The first was the polling rate and making sure that we were in fact sampling at the correct frequency when starting DMA. Due to our sampling rate being 10kHz, and to our buffer having 10000 entries, our total time between HAL_ADC_Start_DMA and the callback of the HAL_ConvCmplCallback needed to be exactly one second. This was confirmed via the blinking red LED on board which indicates that the ADC is sampling, and more accurately with the Keil timer in the debugger to accurately time the period between the two functions mentioned above. The second aspect that needed to be tested was the actual microphone data and see if the values recorded can be trusted and used. Our first test was to connect the ADC input pin to the 5V pin of the Discovery board. The values expected to be read would be the maximum 12 bit value that our ADC could convert, $2^{12} = 4095$. Once confirmed that the ADC value would change we connected the microphone and captured some test measurements of 1 second samples of sound. The following figure shows the captured sample of the spoken number .

Figure 10: Captured 10kHz sample of the spoken digit seven



With testing at different sampling rates, we decided that 10kHz was the lowest we would be willing to go so that the speech API on the hosted web page end of the pipeline would be able to accurately recognize what the spoken digit was. We also noted that most captured samples varied greatly in the amount of noise they produced, and were we to have more time, a digital or analog low pass filter prior to running our speech through the speech API would be needed to increase the accuracy of our number recognition.

4.2 Accelerometer

Our accelerometer component was tested in a similar way compared to the microphone. The first test we ran was to see if our watch variables in Keil would be updated at the right frequency and display reasonable values for pitch and roll when our board is flat on a surface. As seen in Figure 9, or final test to confirm if our accelerometer data was really valid was to display it on our hosted web page. The resulting data is due to the user rotating the discovery board and then setting it down. Because the board was propped up on one end to allow for easier threshold detecting when tapping and double tapping, the resting pitch is 0 but the roll is negative due to the board's angle. With our data making sense, we decided that having LEDs display the intermediate state of the FSM's First Tap state (Seen in Figure 4) would give good user feedback for how fast they needed to double tap and if the board had even detected their first tap. With our board propped on a surface with more give and bounce, coupled with the LED user feedback of the tap detection, our accelerometer was working most of the time and with rare detection misses.

4.3 UART

UART transmission between the Nucleo and Discovery was simple and easy to test. With one of the devices waiting on a UART_Receive with a MAX_Timeout delay and the other transmitting we would be able to watch the receiving buffer fill up when the interrupt was triggered from having received the set amounts of bits from the transmitter. Through our numerous tests, we noticed that data transmission was very reliable as long as the receiving end managed to enter their UART_Receive state before the sender ever starts transmitting data.

4.4 BLE

This was the more difficult component of the pipeline to test as we had issues with the transmission of the BLE samples from the Nucleo to the Android. We tested this component by sending accelerometer pitch and roll data over BLE as at the time of testing our Discovery and Nucleo setup was fully implemented to do so. The only way to confirm the transmission of the sample was through the Android logs discussed in the following section. From the Nucleo end there was no confirmation or handshake to depend on and therefor made us move on to the next component to test, the Android application.

4.5 Android Application

The android testing required us to be able to connect to the Nucleo and receive the microphone and accelerometer GATT characteristics when connecting at the start. This was made easier with Android Device Bridge (ADB) which let us monitor the Android logs as they happened in real time. We were able to see when the connection was being established through the UI of the application, but being able to see the BLE packets coming in with their corresponding characteristics let us know exactly when our application was receiving data.

4.6 Web Server

Testing this last stage of our project was challenging as our microphone and accelerometer data had to make it all the way to the end of the pipeline. Due to this stage's dependencies on prior

components not only working individually, but also working as a whole, we created testing functions that would test the server's ability to accept and process data. This was done with the functions in the api_test.py file where the function test_with_live_recording would replicate the same sampling as our ADC and pass it through to the speech API for processing and the result is subsequently printed. For testing in our full pipeline we ended up tracking the server logs (similar to testing with ADB) as the android connection was being established and data was being transmitted. As a final check prior to the data being either graphed or processed in the speech API, we compared the data transmitted at this point with our known sample points in our buffers on the Nucleo and Discovery to make sure the data was consistent. As discussed in the microphone testing section, we noticed that due to the low quality audio playback of 8kHz sampled audio, and the fact that the API was trained on speech at higher sampling rates, we decided to use a bigger buffer to store the speech on our hardware for the increased performance of our speech recognition in the cloud.

4.7 Entire System

After testing and determining that these sub-components of our pipeline could preform their functionally individually, we moved on to the full scale pipeline testing. The simplest operation to test was the accelerometer as it only needed to perform a one way communication link with the web server. It was also not as much of a strain on the BLE link as less data was being transmitted to the Android from the Nucleo. Figure 9 is the web server's resulting UI from a successful full pipeline transmission of the accelerometer data. We encountered more difficulty in sending the microphone data over BLE due to the fact that there simply were more samples to be lost. Lost samples hindered our speech API's performance which clearly worked with the individual testing of the web server's component. Our weak link was the BLE transmission but overall the rest of our system functioned properly. When we force the speech API to return a value for the number of times the discovery should blink the on board LED we were successfully able to get the right result. Overall, our system individually preformed up to part with the project specification with a slight issue in the microphone BLE transmission that caused our API to not be able to fully recognize the spoken digit from the discovery component.

5 Conclusion

To conclude, the authors wish to point out that their quest for providing the Discovery Board with the opportunity to communicate data over the vast Internet was successful. Despite having no ability to communicate wirelessly at all, given the design of the other components of the system, such as the Nucleo Board Middleman and the Android Application Prophet, it was possible to create a pipeline from the Discovery's UART pins all the way to the Flask Web Server's HTTP endpoints.

This result has a considerable impact on wireless technology and the state of microcontrollers. Due to the relatively low cost of microcontrollers such as the STM32F407 Discovery and the STM32F401RE Nucleo, creating application-specific systems for wireless communication is both affordable and feasible. Furthermore, when designed properly, the Discovery and Nucleo boards should be particularly energy-efficient, which is yet another attractive feature. Moreover, the small form factor of these boards allows them to be particularly portable and mobile.

The authors wish to pose the lessons learned in the design of this system as an allegory to the tale of the four sons in the *Haggadah*. The four sons are described as follows:

1. The wise son
2. The wicked son
3. The simple son
4. The son who does not know how to ask a question

Part of the fascination of the story stems from the idea that these sons are characterized this way. Is everything so black and white? Is the wise son smart, or is he just a smart aleck? In relation to this system, it is not necessarily easy to determine which component is "the wise son". It may seem as though it's the web server, as it has likely the most powerful computational resources. But does being given resources make one wise? One may consider the wicked son to be the Nucleo Board, due to the sheer frustration caused by configuring BLE and UART simultaneously on it. But by causing frustration, does that make one wicked? Let's not forget the good that the Nucleo brought, that being the ability to bridge the gap between the Discovery and the Android Application. The Android Application may seem like the simple son, due to its ease of use relative to its immense power, though is it not possible that it's ease of use is a byproduct of very challenging development behind the scenes? And finally, one may assume the Discovery Board to correspond to the son that does not know how to ask a question. However, there is a very important difference between *not knowing* how to ask a question, and being *unwilling* to ask a question.

As in the case of the famous story from the *Haggadah*, it becomes difficult to characterize the four components in such a way. The ultimate conclusion from the story is that, in fact, this task is impossible. Each son can be wise, each son can be wicked, each son can be simple, and each son may refrain from asking questions – these characteristics are not mutually exclusive. And this, the authors believe, is the most profound concept that was demonstrated over the course of this project. The Discovery Board, with so few resources, is literally incapable of communicating over WiFi. However, it was shown that this *does not prohibit it from doing so*. Ninety-nine people out of a hundred likely would have classified the Discovery Board as "the son who could not ask a question", and yet in reality, with a little bit of clever design, this notion was clearly refuted.

Appendices

Appendix A - Setting up the Web Server

Some minor setup is necessary in order to run the web server, however the process is drastically simplified when taking advantage of some serendipitous Python tools. Firstly, Python 3 will be required. Furthermore, it is recommended to use `virtualenv` to keep dependencies managed. Finally, this tutorial will be assuming the user has access to the free open-source Python package manager, `pip`.

Firstly, navigate to the `api/` directory in this project's root. It is strongly recommended to create a `virtualenv` here, but it is not necessary. Next, execute the following command:

```
$ pip install -r requirements.txt
```

This will install all Python package dependencies required for the project. With these dependencies installed, the server is ready to run! Execute the following command, still within the `api/` directory:

```
$ python api.py
```

The command above launches the web server on port 5000. The server will run indefinitely. When it is desired to kill the server, navigate to the shell through which the server is being run and press `CTRL + C`.

Note that for the end-to-end pipeline described in this paper, the Android application must be configured to send HTTP packets to the IP of the machine running this server. In order to do this, navigate to the file at:
`<project root>/android/app/src/main/java/com/ecse426/project/app/AppController.java`

In this file, edit the string `WEBSITE_ADDRESS` to contain the IP address of the machine running the server. For the changes to take effect, the Android app must be rebuilt and reinstalled on an Android device.

Appendix B - Setting up the Android Application

Some setup is required to run the Android application since it is not available on the Google Play Store.
The user has to download and install the latest version of Android Studio. Once the IDE has been installed, the user must clone the source code from this repository.
Once the source code has been downloaded and the IDE installed, the user must `Import` the Android project by starting Android Studio and selecting the `Open existing Android project` within the Project Menu. From there, Android Studio prompts you with a File Explorer. Within the File Explorer, navigate to the source code that has just been dowloaded and under MicroP/project/android, select the `build.gradle` file. Android Studio will import the project and automatically start downloading any dependencies required for the project.
Once that is complete, connect your Android phone via USB to your computer. Now within Android Studio, click on either `Run > Run` to install the application on your phone or on `Run > Debug` to install and launch a debug session for the application, allowing the user to view any logs within the console.
With either choice, the Android application is now installed and ready to use!

Work Breakdown

In general, most students contributed to a variety of different tasks over the course of the project. A brief summary of the tasks that each student contributed to is given in the subsections below.

Harley Wiltzer

With regard to the design of the system Harley's contributions mainly pertained to the design of the Discovery Board software and the Flask Web Server. Harley originally set up the accelerometer on the Discovery Board and designed the tap detection system. He also designed the filters used on the accelerometer data. Furthermore, he contributed to the implementation of UART communication between the Discovery and Nucleo Boards.

In terms of the web development portion, Harley designed the API endpoint used for the accelerometer data, including a method of parsing the data from the `POST` request, storing the data as a CSV file on the server and generating a graph of the data. Moreover, this included the design of the web page used as the user interface.

Furthermore, Harley contributed to resolving a bug that prevented BLE communication from the Nucleo to the Android Application. This was a massive bug that set the team back many days, and it was resolved through extensive debugging and testing.

Finally, Harley completed the implementation of the WiFi communication between the Android Application and the web server, after older implementations had been made obsolete due to major changes in the system's design.

Matthew Lesko-Krleza

Matthew's major contributions were towards the smart phone android application's implementation, testing and integration. He implemented the core features of the application which involved: scanning for BLE peripheral devices, enabling a GATT connection between client and server devices, the initial code for reading/writing data from/to the Nucleo board, the initial code for saving data to a file on the smartphone, WiFi communication with the web server, as well as the user interface for the application. Since the smart phone is an intermediary component, this required him to test and debug, and modify the android application such that it integrates properly with the Nucleo board and the web server.
Moreover, he wrote code for custom BLE services and characteristics on the Nucleo board, but were later removed due to system design changes.
Overall he had to learn about the BLE stack, smartphone WiFi communication, and how the Nucleo board sends data over BLE.

Fabrice Normandin

Fabrice contributed by writing software, including major portions of the Discovery and Nucleo board's programs, as well as developing a custom Speech Recognition model in TensorFlow (which was unfortunately not used), the Speech and initial Accelerometer endpoints, the interfacing with the Google Speech API, and the refactoring of the Android application code.

Marcel Morin

Marcel's contributions were mainly in the hardware domain of this project. Getting UART and the microphone data via DMA on the board was what he worked on early in the project. He also helped setup the Nucleo interrupts that would indicate what kind of data was to be transmitted from the Discovery. Once the reliable communication between the Discovery and Nucleo board was establish (due to UART) Marcel tested the full design pipeline to see the performance of the data transmission between all components of the pipeline.